APPENDIX A

The following is C language source code to implement transactional optimization based on observable state using new value logging (NVL). (It is also possible to implement this optimization using old value logging or other transactional mechanisms.)

```
/*
 * nvl_tran.c
 * transaction logging mechanism
 *
 * This transaction mirror is initially designed for BYTE and SHORT atomic updates.
 * Because this type of mirror request a logged values scan for each read access,
 * it relies on read access drivers to be implemented.
 *
 * Each time a java method makes more than 1 atomic write (BYTE or SHORT)
 * the transaction mechanism is advantageous, but it has to be under certain conditions.
 * Writings must occurs betweens 2 flush points which are:
 *  - observable state points, such as communications
 *  - System limitations, such as Util.array_xxx methods or certain other native methods
 *   (Crypto for ex) that rely on validity of java byte/short/arrays.
 *
 */

#include <gpos.h>
#include <nvl_tran.h>

#if _include_NEW_VALUE_LOGIN_MIRROR == ON

#define SDBG(x)         //x      /* statistical information display */

SDBG(static BYTE _SXDATA maxNVLVolatileEntriesCount = 0;)
SDBG(static WORD _SXDATA doubleAccessEntry = 0;)
SDBG(static WORD _SXDATA eepCumulAdvantage = 0;)

/* EEPROM NVL mirror */
#ifdef _WIN32
        #define NVLPersistentMirror             SYS_Persistent.NVL_PersistentMirror
#else
        NVL_MIRROR _SXDATA NVLPersistentMirror;
#endif

#define NVLPersistentCrc                NVLPersistentMirror.crc
#define NVLPersistentEntriesCount NVLPersistentMirror.count
#define NVLPersistentEntries            NVLPersistentMirror.entries

/* RAM NVL mirror */
#define NVLVolatileMirror               SYS_VolatileBuffer.NVL_VolatileMirror
#define NVLVolatileCrc                  NVLVolatileMirror.crc
#define NVLVolatileEntriesCount         NVLVolatileMirror.count
#define NVLVolatileEntries              NVLVolatileMirror.entries

/*
        NVL_ScanEntries
        parse the volatile Logging mirror and return the entry value,
        if an entry matches the input address.
*/
E_ADDRESS NVL_ScanEntries(E_ADDRESS addr)
{
        BYTE iEntry;

        for (iEntry = 0; iEntry < NVLVolatileEntriesCount; iEntry++) {
                if (VIRTUAL_ADDRESS(NVLVolatileEntries[iEntry].dest) == addr) {

                        /* an entry matches the address, return the indirect one */
                        switch (NVLVolatileEntries[iEntry].type) {
```

```
                                        case T_BYTE:
                                                return
(E_ADDRESS)(&NVLVolatileEntries[iEntry].value.bw.low);
                                                break;
                                        case T_SHORT:
                                                return
(E_ADDRESS)(&NVLVolatileEntries[iEntry].value);
                                                break;
                                }
                                /* we should never arrive here */
                                SC_ASSERT(FALSE);
                        }
                }
                /* no entry matches the address, return the original one */
                return addr;
}

/*
        NVL_FlushReset
        Check validity of the persistent mirror thanks its crc.
        Flush value entries from the persistent mirror to their destinations.
        Reset persistent and volatile mirror once operation is over.

        Note: if a tear occurs during reset, invalidity of crc will help us catch it.
*/
void NVL_FlushReset()
{
        BYTE iEntry;
        WORD wLength;

        if (NVLPersistentEntriesCount > 0) {

                /* it is possible that a tearing occurs during the persistent buffer update,
                        hence NVLPersistentEntriesCount could be a meaningless value. If it is
                        the case, the entire buffer is reset */
                if (NVLPersistentEntriesCount > MAX_NLV_ENTRIES) {
                        wLength = sizeof(NVL_MIRROR);
                } else {
                        //wLength = offsetof(NVL_MIRROR,entries[NVLPersistentEntriesCount]);

                        wLength = sizeof(BYTE) + sizeof(WORD) + (NVLPersistentEntriesCount *
sizeof(NVL_ENTRY));
                }

                if
(_OS_ComputeCRC16(VIRTUAL_INDEX(&NVLPersistentEntriesCount),(WORD)(wLength -
sizeof(WORD))) == NVLPersistentCrc) {
                        for (iEntry = 0; iEntry < NVLPersistentEntriesCount; iEntry++) {
                                switch (NVLPersistentEntries[iEntry].type) {
                                        case T_BYTE:

        _OS_WriteMemory((GEN_ADDRESS)&NVLPersistentEntries[iEntry].value.bw.low,
NVLPersistentEntries[iEntry].dest,(WORD)sizeof(BYTE));
                                                break;
                                        case T_SHORT:

        _OS_WriteMemory((GEN_ADDRESS)&NVLPersistentEntries[iEntry].value,
NVLPersistentEntries[iEntry].dest,(WORD)sizeof(WORD));
                                                break;
                                }
                        }
                }

                /* clear the NVL Volatile buffer (crc + count + n*entries) */
                _OS_ClearRAMBuffer((GEN_ADDRESS)&NVLVolatileMirror, wLength);
                /* reset the NVL Persistent Buffer (crc + count + n*entries) */
```

```
                    _OS_WriteMemory((GEN_ADDRESS)&NVLVolatileMirror,
VIRTUAL_INDEX(&NVLPersistentMirror), wLength);

        }
}


/*
        NVL_SaveFlushReset
        Backup the volatile Logging mirror into its persistent image.
        Call the FlushReset function once operation is over.
*/
void NVL_SaveFlushReset()
{
        WORD wLength;

SDBG(
                fprintf(stderr, "flush NVL mirror : number of entries = %x\n",NVLVolatileEntriesCount);
                if (NVLVolatileEntriesCount > maxNVLVolatileEntriesCount)
maxNVLVolatileEntriesCount = NVLVolatileEntriesCount;
                fprintf(stderr, "Current Max number of entries = %x\n",maxNVLVolatileEntriesCount);
                fprintf(stderr, "number of already present entry NVL access =
%x\n",doubleAccessEntry);

                if (NVLVolatileEntriesCount > 0) {
                        eepCumulAdvantage += ((NVLVolatileEntriesCount * 3) - (1 /* saving to EEP
buff */ + 1 /* reseting EEP buff */ + NVLVolatileEntriesCount /* number of flushes */));
                }
                fprintf(stderr, "number EEPROM writing saved thanks NVL =
%x\n",eepCumulAdvantage);
                fprintf(stderr, "total number EEPROM writing saved = %x\n",eepCumulAdvantage +
doubleAccessEntry);

)

        if (NVLVolatileEntriesCount > 0) {
                /* working length */
                //wLength = offsetof(NVL_MIRROR,entries[NVLVolatileEntriesCount]);
                wLength = sizeof(BYTE) + sizeof(WORD) + (NVLVolatileEntriesCount *
sizeof(NVL_ENTRY));
                /* compute crc */
                NVLVolatileCrc =
_OS_ComputeCRC16(VIRTUAL_INDEX(&NVLVolatileEntriesCount),(WORD)(wLength -
sizeof(WORD)));
                /* backup to the persistent image */

        _OS_WriteMemory((GEN_ADDRESS)&NVLVolatileMirror,VIRTUAL_INDEX(&NVLPersistentMi
rror),wLength);
                /* flush the persistent mirror & reset both persistent and volatile buffers */
                NVL_FlushReset();
        }
}


/*
        NVL_LogEntry
        Add an entry in the Logging volatile mirror, if entry is not already existing.
        If volatile mirror is getting full: backup,flush and reset it.
*/
void NVL_LogEntry(INDEX dest, BYTE type, WORD value)
{
        BYTE iEntry;

        /* check if an entry is already present is the Volatile Buffer */
        for (iEntry = 0; iEntry < NVLVolatileEntriesCount; iEntry++) {
                if (NVLVolatileEntries[iEntry].dest == dest)  {
                        SDBG(doubleAccessEntry++;)
```

```
                        break;
                }
        }

        NVLVolatileEntries[iEntry].dest = dest;
        NVLVolatileEntries[iEntry].type = type;
        NVLVolatileEntries[iEntry].value.w = value;

        if (NVLVolatileEntriesCount == iEntry) {
                NVLVolatileEntriesCount++;
                /* check if there is enough place for another entry, if not save and flush */
                if (NVLVolatileEntriesCount == MAX_NLV_ENTRIES) {
                        NVL_SaveFlushReset();
                }
        }

}

/*
 * _VM_WriteByteSecu()
 *
 * Writes a byte in Eeprom at address specified
 *
 */
void _VM_WriteByteSecu(INDEX iaddress,BYTE bvalue)
{
        DBG(fprintf(stderr, "Write BYTE through NVL mirror\n");)

        if (IS_SETFLAG2(OS_TransactionActive)) {
                _OS_WriteByteSecu(iaddress, bvalue);
        } else {
                if (IS_EEPROM(iaddress)) {
                        NVL_LogEntry(iaddress,T_BYTE,(WORD)bvalue);
                } else {
                        VIRTUAL_ADDRESS(iaddress)[0] = bvalue;
                }
        }
}

/*
 * _VM_WriteWordSecu()
 *
 * Writes a word in Eeprom at address specified
 *
 */
void _VM_WriteWordSecu(INDEX iaddress,WORD wvalue)
{
        DBG(fprintf(stderr, "Write WORD through NVL mirror\n");)

        if (IS_SETFLAG2(OS_TransactionActive)) {
                _OS_WriteWordSecu(iaddress, wvalue);
        } else {
                if (IS_EEPROM(iaddress)) {
                        NVL_LogEntry(iaddress,T_SHORT,wvalue);
                } else {
                        (*(WORD*)(VIRTUAL_ADDRESS(iaddress))) = (WORD)wvalue;

                }
        }
}

#endif
```